Module 4: Understanding State Management

Terraform level 1 - Day 1 - **Module 4** (SSM Version)

Module 4: Learning Objectives

By the end of this module you will understand:

- What Terraform state is and why it's essential
- How state maps your configuration to real AWS resources
- Why state files contain sensitive data and must be protected
- How to inspect and understand your state file
- The limitations of local state for team collaboration
- How state locking prevents concurrent modification conflicts

Recap: What We've Learned

- Module 1: Terraform is declarative IaC - you describe what you want, Terraform figures out how to build it
- Module 2: HCL syntax - providers, resources, data sources, arguments, and blocks Module 3: Core workflow - init, validate, plan, apply, destroy
- Now in Module 4: Let's understand how Terraform remembers what it created

What is State?

State is Terraform's memory - a record of all resources it manages

Think of it like a bank's asset inventory system:

- Each IT asset (server, database, network) is catalogued with its unique ID
- Current attributes are recorded (location, configuration, status)
- Without the inventory, you wouldn't know what infrastructure you own or where it is

Without state: Terraform wouldn't know:

- What resources it already created
- What needs to be updated vs. created
- What can be safely deleted

Why State Matters

Remember from Module 1: Terraform is idempotent - running apply multiple times produces the same result

How does Terraform achieve this? State!

The magic trick:

1. You write configuration (desired state)
2. State file tracks current reality
3. `terraform plan` compares them
4. `terraform apply` makes reality match your configuration

Exercise Overview

In this module we'll:

- Create a secure parameter in AWS using SSM Parameter Store
- Explore how Terraform tracks resources in state
- Discover the critical security lesson about state files

The approach: We'll use a hardcoded password for simplicity and learning purposes

The critical lesson: State files contain sensitive data and must be protected!

Exercise: Setup

We'll create a secure parameter in AWS and explore state

Step 1: Create a new project directory

```
cd day1/module4
mkdir module4-project-ssm
cd module4-project-ssm
```

Step 2: Create the provider configuration file

Create `terraform.tf` :

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 6.20.0"
    }
  }
  required_version = ">= 1.3.5"
}
```

Exercise: Initialize Project

Step 3: Initialize the Terraform project

```
terraform init
```

What happened?

- Downloaded the AWS provider plugin
- Created `.terraform/` directory for plugins
- Created `.terraform.lock.hcl` to lock provider version

Step 4: Check what files exist (no state file yet!)

```
ls -la
```

Notice: No `terraform.tfstate` file yet - we haven't created any resources!

Banking Scenario: You need to store a database password securely for a development application.

Step 5: Create `main.tf` with provider and parameter configuration

```
provider "aws" {
  region = "us-east-1"

  # IMPORTANT: Change 'userX' to your assigned student ID (user1, user2, etc.)
  default_tags {
    tags = {
      owner = "userX"
    }
  }
}

resource "aws_ssm_parameter" "db_password" {
  name        = "/dev/database/master-password"
  description = "Master password for development database"
  type        = "SecureString"   # Encrypted in AWS!
  value       = "TempPassword123!"   # DEMO ONLY - never do this in production!
}
```

Important: About This Example

This example intentionally uses a hardcoded password for learning purposes

In production, NEVER:

- Hardcode passwords in configuration
- Use weak passwords
- Skip using generated passwords or AWS Secrets Manager

For this training: We're using a simple hardcoded password to demonstrate how sensitive data appears in state files.

Exercise: Validate Configuration

Step 6: Check for syntax errors

```
terraform validate
```

You should see: "Success! The configuration is valid."

Step 7: Plan the changes

```
terraform plan
```

What to observe:

- Terraform shows it will CREATE a new SSM parameter
- Lists all the attributes that will be set
- No existing state to compare against (first run)

Exercise: Create the Parameter

Step 8: Apply the configuration

```
terraform apply
```

Type `yes` when prompted.

What's happening:

- Terraform calls AWS API to create SSM parameter
- AWS stores it encrypted with KMS (SecureString type)
- AWS generates a unique ARN for the parameter
- Terraform will save all this information to state

Exercise: The State File Appears!

Step 9: Once apply completes, check the directory

```
ls -la
```

New files created:

- `terraform.tfstate` - **The state file!**

This is Terraform's memory of what it created

Exercise: Inspect the State File

Step 10: Let's look at the state file (it's JSON)

```
cat terraform.tfstate
```

What you'll see:

- Version information
- Resource type: `aws_ssm_parameter`
- Resource name: `db_password`
- Attributes with actual values from AWS

It's readable, but lots of information!

Understanding State Structure

Key sections in state:

```
{
  "version": 4,
  "terraform_version": "1.13.5",
  "resources": [
    {
      "type": "aws_ssm_parameter",
      "name": "db_password",
      "instances": [
        {
          "attributes": {
            "name": "/dev/database/master-password",
            "type": "SecureString",
            "value": "TempPassword123!",
            "arn": "arn:aws:ssm:us-east-1:..."
          }
        }
      ]
    }
```

Exercise: Better Ways to View State

Step 11: Use Terraform commands instead of reading JSON

List all resources in state:

```
terraform state list
```

Output: `aws_ssm_parameter.db_password`

Show detailed attributes:

```
terraform state show aws_ssm_parameter.db_password
```

Much more readable! Shows all attributes Terraform is tracking.

The Configuration to Reality Mapping

Your Configuration (what you wrote):

```
resource "aws_ssm_parameter" "db_password" {
  name = "/dev/database/master-password-userX"
  type = "SecureString"
  value = "TempPassword123!"
}
```

State File (what Terraform tracks):

```
{
  "name": "/dev/database/master-password",
  "arn": "arn:aws:ssm:us-east-1:123456789:parameter/dev/database/master-password",
  "type": "SecureString",
  "value": "TempPassword123!",
  "version": 1
}
```

Critical Security Issue: Passwords in State

Step 12: Search for sensitive data in state

```
grep -i password terraform.tfstate
```

You'll find: `"value": "TempPassword123!"`

The password is stored in PLAIN TEXT!

Wait... didn't we use **`type = "SecureString"`** **?**

The critical lesson:

- `SecureString` means AWS ENCRYPTS it in Parameter Store
- But Terraform needs to know the ACTUAL VALUE to manage it
- So Terraform stores it in PLAIN TEXT in the state file!

The Secure vs. Insecure Paradox

In AWS: Your password IS secure

```
# Try to get the parameter without decryption
aws ssm get-parameter --name "/dev/database/master-password-userX"
# You can't see the value - it's encrypted!

# You need special permission to decrypt
aws ssm get-parameter --name "/dev/database/master-password-userX" --with-decryption
# Now you can see it - but this requires KMS permissions
```

In Terraform State: Your password is NOT secure

```
cat terraform.tfstate | grep value
# Shows plain text password - no encryption, no special permissions needed!
```

This is why state files must be protected!

What Else is in State?

Step 13: Look for other sensitive or important data

```
terraform state show aws_ssm_parameter.db_password | grep -E "name|arn|value|version"
```

You'll find:

- Parameter name (path in SSM)
- AWS ARN (Amazon Resource Name)
- The actual password value
- Version number

All of this is information that could help an attacker!

Banking Security Context

In real banking infrastructure, state files contain:

- Database passwords
- API keys and secrets
- Network configurations
- Security group rules
- KMS key IDs
- Resource ARNs

If state is compromised = security breach

Production requirements:

- Encrypted state storage (S3 with encryption)

Exercise: Make a Change

Step 14: Update the parameter configuration

Modify `main.tf` - change the description:

```
resource "aws_ssm_parameter" "db_password" {
  name        = "/dev/database/master-password-userX"
  description = "Updated: Master password for dev database"  # Changed!
  type        = "SecureString"
  value       = "TempPassword123!"
}
```

Exercise: See State Comparison

Step 15: Plan the change

```
terraform plan
```

What Terraform does:

1. Reads configuration: `description = "Updated: Master password..."`
2. Reads state: `description = "Master password..."`
3. Detects difference
4. Plans to UPDATE

Output shows:


```
~ description = "Master password for development database" -> "Updated: Master password for dev database"
```

Exercise: Apply the Change

Step 16: Apply the update

```
terraform apply
```

What happens:

1. Terraform modifies the SSM parameter
2. AWS updates the parameter
3. Terraform updates the state file with new values
4. Creates a backup of the old state

Step 17: Verify the state was updated

```
terraform state show aws_ssm_parameter.db_password | grep description
```

Exercise: Check the Backup

Step 18: Look at state backup

```
ls -la terraform.tfstate*
```

You'll see:

- `terraform.tfstate` - current state
- `terraform.tfstate.backup` - previous version

Step 19: Check what's in the backup

```
grep description terraform.tfstate.backup
```

Shows: The old description

Safety net! If something goes wrong, you have the previous state.

Local State Limitations

Now let's discuss team collaboration challenges

Scenario: You and 3 colleagues managing banking infrastructure

Problems with local state:

1. **Where is the truth?**
   - Engineer A has state on their laptop
   - Engineer B has different state on their laptop
   - Who has the correct version?
2. **How do you share?**
   - Email state files? (Security risk!)
   - Commit to Git? (Passwords exposed!)

The Team Collaboration Problem

Real example:

Monday: Engineer A creates 10 SSM parameters, state on their laptop
Tuesday: Engineer B tries to update parameter values
Problem: Engineer B's Terraform doesn't know about those parameters!

Result:

- Errors and conflicts
- Manual coordination required
- Risk of duplicate resources
- Risk of deleting production resources by accident

Solution: Remote state (covered in Day 2, Module 5)

Exercise: State Locking

What is state locking?

- Prevents two people from modifying state simultaneously
- Like a "do not disturb" sign on infrastructure changes
- Prevents state corruption

Let's simulate this:

Exercise: Simulate Concurrent Changes

Step 20: Split your VS Code terminal (View →Terminal →Split Terminal)

Terminal 1: Start an apply (don't confirm yet)

```
terraform apply
```

Wait at the prompt - don't type "yes" yet

Terminal 2: While Terminal 1 is waiting, try to apply

```
terraform apply
```

Understanding the Lock Error

In Terminal 2, you'll see:

```
Error: Error acquiring the state lock

Error message: resource temporarily unavailable
Lock Info:
  ID:        c8740628-c470-c3d7-c7ed-9e627cf13aa6
  Path:      terraform.tfstate
  Operation: OperationTypeApply
  Who:       your-username@your-machine
  Version:   1.13.5
  Created:   2025-11-15 14:23:45

Terraform acquires a state lock to protect the state from being
written by multiple users at the same time.
```

This is GOOD! The lock is protecting your state.

Exercise: Release the Lock

Step 21: In Terminal 1

- Type "no" to cancel the apply (we don't actually want to make changes)
- The lock is released

Step 22: In Terminal 2, try again

```
terraform apply
```

It works now! Type "no" to cancel.

Key lesson: Locks prevent simultaneous modifications that could corrupt state.

How State Locking Works

The lock lifecycle:

1. Engineer A runs `terraform apply`
2. Terraform creates a lock file: `.terraform.tfstate.lock.info`
3. Engineer B tries to run `terraform apply`
4. Terraform sees the lock file and refuses to proceed
5. Engineer A's operation completes
6. Lock file is deleted
7. Engineer B can now proceed

Local locking: Uses file system locks (simple but limited)

Remote locking: Uses DynamoDB (for S3 backend) - much more robust

Exercise: Useful State Commands

Let's practice working with state: List

all resources:

```
terraform state list
```

Show specific resource:

```
terraform state show aws_ssm_parameter.db_password
```

Discussion: State Management Challenges

Based on what we've learned, discuss:

- What happens if the state file is lost or corrupted?
- How would you share state across a team of 10 engineers?
- What about state for production vs. development environments?
- How do we protect passwords and secrets in state?

Real-World Banking Example

Large Bank Infrastructure Team:

- 50 engineers across 5 teams
- Managing 1000+ AWS resources
- Multiple environments (dev, test, staging, prod)
- Strict compliance and audit requirements

With local state: Impossible to coordinate

With remote state (Day 2):

- Single source of truth in encrypted S3 bucket
- State locking via DynamoDB prevents conflicts
- Version history for audit trail

Exercise: Simulating State File Issues

Understanding what happens when state goes wrong

State files are critical to Terraform's operation. Let's explore what happens when:

1. State gets corrupted (manual edits)
2. State gets lost (deleted file)

Warning: This is for learning purposes only! Never do this in production!

Scenario A: Corrupted State

Step 23a: First, let's backup our current working state

```
cp terraform.tfstate terraform.tfstate.safe_backup
```

Step 23b: Manually corrupt the state file

Open `terraform.tfstate` in your editor and find the parameter name. Change it from:

```
"name": "/dev/database/master-password-userX",
```

to:

```
"name": "/dev/database/CORRUPTED-password",
```

Save the file.

Observe: Terraform Detects the Corruption

Step 23c: Now run a plan

```
terraform plan
```

What you'll see:

```
Terraform will perform the following actions:

  # aws_ssm_parameter.db_password must be replaced
-/+ resource "aws_ssm_parameter" "db_password" {
      ~ name = "/dev/database/CORRUPTED-password" -> "/dev/database/master-password-userX" # forces replacement
```

What's happening:

- State says the parameter is named `/dev/database/CORRUPTED-password`
- AWS reality says no such parameter exists
- Configuration says it should be `/dev/database/master-password`

Understanding the Corruption Impact

The problem:

- Terraform's state no longer matches AWS reality
- Terraform will try to destroy a non-existent resource
- Then create what it thinks is a new resource (but already exists!)
- This could cause errors or duplicate resources

This is why you NEVER manually edit state files!

Restore from Backup

Step 23d: Restore the good state

```
cp terraform.tfstate.safe_backup terraform.tfstate
```

Step 23e: Verify it's fixed

```
terraform plan
```

You should see: "No changes. Your infrastructure matches the configuration."

Lesson: Always keep backups! Terraform creates `.backup` files automatically.

Scenario B: Lost/Deleted State

Now let's see what happens when state is completely lost Step

24a: "Lose" the state file (we'll hide it)

```
mv terraform.tfstate terraform.tfstate.hidden
mv terraform.tfstate.backup terraform.tfstate.backup.hidden
```

Step 24b: Check what files exist

```
ls -la terraform.tfstate*
```

You should see: Only `terraform.tfstate.safe_backup` remains

Observe: Terraform Loses Its Memory

Step 24c: Run a plan without state

```
terraform plan
```

What you'll see:

```
Terraform will perform the following actions:

  # aws_ssm_parameter.db_password will be created
  + resource "aws_ssm_parameter" "db_password" {
      + name  = "/dev/database/master-password-userX"
      + type  = "SecureString"
      + value = (sensitive value)
      ...
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

Understanding Lost State Impact

What's happening:

- No state file exists
- Terraform has amnesia - doesn't remember creating the parameter
- Terraform thinks it needs to CREATE the resource
- But the resource ALREADY EXISTS in AWS!

What would happen if you applied?

```
terraform apply
```

Terraform will try to create a new resource,
but resource already exists, so creation will fail.

Restore the State

Step 24d: Bring back the state file

```
mv terraform.tfstate.hidden terraform.tfstate
mv terraform.tfstate.backup.hidden terraform.tfstate.backup
```

Step 24e: Verify everything is back to normal

```
terraform plan
```

You should see: "No changes. Your infrastructure matches the configuration."

Terraform's memory is restored!

Key Lessons from State Corruption

What we learned:

1. **Corrupted state** causes Terraform to misunderstand reality
   - Leads to incorrect plans
   - Could destroy/recreate resources unnecessarily
   - Never manually edit state files!
2. **Lost state** causes Terraform amnesia
   - Terraform forgets what it manages
   - Tries to recreate existing resources
   - Creates conflicts and errors
   - Catastrophic in production!

Exercise: Clean Up

Step 25: Destroy the resources

```
terraform destroy
```

Type `yes` to confirm.

What happens:

1. Terraform reads state file
2. Identifies resources to delete
3. Calls AWS API to delete SSM parameter
4. Updates state file to reflect deletion
5. State file remains but resources list is empty

Exercise: Check Final State

Step 24: After destroy completes, check the state

```
cat terraform.tfstate
```

You'll see:

```
{
  "version": 4,
  "terraform_version": "1.13.5",
  "resources": []
}
```

Empty resources array! Terraform knows it's not managing anything now. **State file still exists** - it remembers the workspace exists, just no resources in it.

Key Takeaways

State is Terraform's memory - maps configuration names to real AWS resource IDs

State enables idempotency - Terraform knows what exists and what needs to change

State contains sensitive data - passwords, API keys, secrets in plain text

- Even when using `type = "SecureString"`
- Even when following best practices!

Local state limitations:

- Not suitable for teams
- Risk of loss (laptop failure)
- No centralized access control
- Contains sensitive data in plain text

Knowledge Check 1

What is the primary purpose of the Terraform state file?

A) To store your .tf configuration files

B) To map configuration resource names to real AWS resource IDs

C) To backup your AWS resources

D) To make Terraform run faster

Knowledge Check 2

- Why does the password appear in plain text in terraform.tfstate even when using type
- = "SecureString"?

A) It's a bug in Terraform

B) You need to enable encryption in the AWS provider

C) Terraform needs to know the actual value to detect changes and manage resources

D) SecureString only works with remote state

Knowledge Check 3

What happens when two engineers try to run terraform apply simultaneously on the same local state file?

A) Both operations succeed and create duplicate resources
B) The second operation is blocked by state locking
C) Terraform automatically merges the changes
D) The state file gets corrupted

What We Built Together

Hands-on accomplishments:

1. Created a new Terraform project from scratch
2. Provisioned AWS resources and watched state file appear
3. Inspected state with both JSON and Terraform commands
4. **Discovered sensitive data in plain text state** - even "SecureString" parameters show passwords
5. Made changes and watched state update automatically
6. Simulated state locking with concurrent operations
7. Cleaned up resources and observed empty state

Critical lesson: State files contain sensitive data and must be protected!

Next: Day 2 Module 5 - Remote State with S3 and DynamoDB!